Proceedings of ICAR 2003
The 11th International Conference on Advanced Robotics
Coimbra, Portugal, June 30 - July 3, 2003

A Development Approach to Industrial Robots Programming

G.V.Arnold
Dpto of Informatics
Campus de Gualtar
Universidade do Minho
4710-057, Braga, Portugal
gva@di.uminho.pt

P.R. Henriques
Dpto of Informatics
Campus de Gualtar
Universidade do Minho
4710-057, Braga, Portugal
prh@di.uminho.pt

J.C.Fonseca
Dpto of Industrial Electronics
Campus de Azurém
Universidade do Minho
4800, Guimarães, Portugal
jaime.fonseca@dei.uminho.pt

Abstract

This paper proposes a development approach to industrial robot programming, that includes: a truly high level and declarative language; an easy-to-use front-end; an intermediate representation; an automatic generator of the robot code generators. So, we introduce a new paradigm to program industrial robots, that focus on the modeling of the system, rather than on the robot. It will improve the programming and maintenance tasks, allowing the reuse of source code, because this source code will be machine independent.

1 Introduction

Today, the programming task of non-robotic systems is done in very high level languages; each of these languages try to facilitate the specification of the program that solves the problem, allowing the programmer to concentrate on the problem, instead of the equipment where this program will be executed.

To do this, the developer should, first, do not think about the equipment that will run the program. He must be concerned about the problem, to find the correct solution for it. To make this task easy, it is used some modeling technique, appropriate for the kind of problem. The modeling technique will decompose the problem into smaller components, that will be implemented using the adequate programming language. This language should facilitate not only the programming task, but the readability, maintenance, reusability, composability and other important features for the development according to software engineering principles.

Another advantage of this approach is the existence of compilers to translate the high-level (machine independent) languages for different computer platforms. With these, it is possible to use the same source code in different machines. So, the equipment can be upgraded, or changed, without the necessity of rewriting the programs.

However, the development of industrial robotic systems is still a difficult, costly, and time consuming operation. Today's industrial robots generally require a tremendous amount of programming to make them useful. Their controllers are not very sophisticated and the commercial robot programming environments are typically closed systems. The manipulator level is still the most widely used programming method employed in industry for manufacturing tasks. The forerunner languages, such as AML [16] or AL [9], have now been superseded by elaborated robot languages like ABB Rapid [1]. Despite their common use, they have three important drawbacks.

1. They require detailed description of the motion of every joint in the mechanism in order to execute a desired movement.
2. They require specialized knowledge of the language.
3. The robot programs have limited portability. As a result, significant investment must be made when changing or acquiring a new robot type or simply when upgrading a new controller from the same vendor.

One simple approach to solve some limitations described above are the Off-line programming environments. These environments are based in graphical simulation platforms, in which the programming and execution process are shown using models of the real objects. Consequently, the robot programmer has to learn only the simulation language and not any of the robot programming languages. Other benefits of off-line programming environments include libraries of pre-defined high-level commands for certain types of applications, such as painting or welding, and the possibility to assess the kinematics feasibility of a move, thus enabling the user to plan collision-free paths. The simulation may also be used to determine the cycle time for a sequence of movements. These environments usually provide a set of primitives commonly used by various robot vendors, and produce

a sequence of robot manipulator language primitives such as "move" or "open gripper" that are then downloaded in the respective robot controllers. However, the current state-of-the-art off-line systems suffer from two main drawbacks. *Firstly*, they do not address the issue of sensor-guided robot actions. *Secondly*, they are limited to a robot motion simulator, which provides no advanced reasoning functionality, nor flexibility in the tasks.

So, we propose an integrated, formal and high-level approach to industrial robot programming, that would solve the above problems. To use this approach, it is necessary to have the following components (some may exist, some others need to be developed):

- a truly high level and declarative language
- an easy-to-use front-end
- an intermediate representation
- an automatic generator of the robot code generators

In the following section, and before describing in more detail our approach (sec. 3), we present (sec. 2) the importance of a modeling technique and discuss the four components of this approach. At the end (sec. 4), appear the conclusions and future works.

2 Background

On the following subsections, the components of our approach are described. The first three subsections deal with to topics relevant to create a simple and friendly interface between the programmer and the compiler. The other two subsections are concerned with matters that make possible the generation of programs that will be executed on different robots.

2.1 Modeling Techniques

The use of an adequate modeling technique will facilitate the development of a programming system, enabling the system developers and the system clients to express their ideas, allowing their communication in a known way. The advantage of modeling is the creation of models from the system and its behavior that can be seen in different abstraction levels, before implementing it. So, it is very important to model the system first.

If the programs are created directly, thinking on the problem and on the machine, this programs would be difficulty to write, to read, and consequently, to maintain. So, some techniques were created, like the structure analysis, that was the first modeling technique (defined in the 70's), where the problem is decomposed based on the data and the operations, that should be modeled separately. To model the data it is used the Entity Relationship Diagram (ERD), and to model the operations it is used the Data Flow Diagram (DFD). Today, there are some others modeling techniques, like Unified Modeling Language (UML), used to model object orienting systems. It is used the Class Diagram, that shows the classes and their relationships in a logical view (like ERD to structure analysis); the State Transition Diagram, that shows the events that causes transition from one state to another, with its resulting actions (like DFD to structure analysis); and the Use-Cases Diagram, that shows the system's use cases and the actors that interact to them.

The robot programming also have its modeling techniques. One modeling technique used in the development of mobile robots, the Subsumption Architecture [14], was used to model a manufacturing cell, composed by two robots and some others components [2]. The subsumption architecture was the first behavior based modeling technique and, even it had been created to develop mobile robots, they can be used, as a high level abstraction, to model industrial applications.

Among various modeling techniques, the analyst should choose the most adequated for a such problem, to obtain the advantages of the software engineering.

2.2 Declarative Language

As it was said before, it is important to use a modeling technique, but is also important to have some language that would allow the programmer to express exactly what he intends to do.

Such a language should be simple, and as closed to the specification of the problem as possible.

For a language to be close to the specification, it must also have high level constructors that allow the definition of structured and complex abstract data types and mathematical operators over them.

There are, basically, two kinds of programming languages:

1. imperative languages: the underlying principle (the operational semantics) is very similar to the processor's execution cycle, being necessary to understand its architecture; the kind of available statements is also similar to the machine instructions. The programmer should also know how to manipulate memory elements to store the necessary data;
2. declarative languages: instead of following the execution principles, those languages have as background a mathematical theory that supports data representation and operations over that data. The explicit memory manipulation is not necessary; the programmer just manipulate, in a high level, the data, without being necessary to know where this data is stored.

Declarative languages are higher level than imperative languages, more closed to the specification of the problem.

One example to show the difference between this two kind of languages is the file manipulation, that can be done in imperative languages (like C) and in declarative languages (like SQL). A simple operation, like searching a file for a specific person (in this example called "John"), in a imperative language is done in the following way:

```
Open file
Read first element
While (the name of the element is not "John",
       and the file did not reach the end)
Do    Read the next element
If John was found, print its data
```

This same operation can be written, in a declarative language, as follows:

```
Select all data from file,
       Where name is equal to "John"
```

According to the style, declarative languages are classified as functional or relational (logic). The first group is supported by the principle that a program is just a function mapping the input data into the output results; while the second family relies upon the idea that a program is a set of assertions defining the relations that hold (evaluate to true) in some world. Typical declarative languages are: Lisp, ML or Haskell (functional paradigm), and Prolog (logic paradigm).

In the context of the robot programming, an example of declarative languages (proposed some years ago [20]) is RS — a real time language relying on the principle of productions systems, ie, on the rule-based paradigm (condition-reaction set of rules). Some experiments where made like controlling a Nachi industrial robot [18], or controlling a simulated manufacturing cell [2].

2.3 Compiler Front-End

After designing the system model and writing its description in a declarative language, the program must be implemented. To do this, there are two ways: the developer writes all the program by hand (maybe applying some systematic translators rules); or he uses a compiler that transforms the specification into a runable program (machine code or still and a high-level language that is then translated into the target code). To facilitate the second approach, it is recommended the use of an environment specially tailored for the application scope of that language, which contains everything necessary for the edition and compilation. This environment should be, also, easy-to-use. So, it must have a friendly interface.

The compiler is normally divided into two components: the front-end (FE) that reads the input and parses it to recognize its meaning (it implements the lexical, syntactic and semantic analysis); and the back-end (BE) that generates the target code and optimizes it.

One well-known description language for industrial automation applications is the Grafcet [7], that is suitable to support a visual interface for the front-end, that is able to produce a textual description from the graphical specification.

2.4 Intermediate Representation

In traditional compilers[19], the interface between FE and BE is an intermediate representation (IR) that should be independent of source and target languages; see for instance the well known RTL language [15].

If the desired independence is reached, it makes possible the generation of programs that will be executed on different robots. The IR is composed by a set of instructions and data representations that is common to the majority of industrial robots. The FE must translate the source program, written by the programmer, into this intermediate representation, and then, after choose the robot, the BE will translate the IR into robot code.

This representation must be as simple as possible to make the code generation easy and efficient.

The IR that will be used is the one proposed in the context of the project Dolphin (proposal submitted to FCT for a national grant under SAPIENS 2002 program), the so called MIR (My Intermediate Representation) that sprang out from the previous BEDS project [13]. This project deals with optimization and code generation tools, aiming to offer a framework to build optimized code generators for different machines based on an universal intermediate program representation; industrial robotics is one of its applications. The universal intermediate program representation proposed is based on previous work on backends generator [13].

2.5 Automatic Generator of Code Generators

An automatic generator of code generators is a program that produces as output a set of routines, that will be included in a new compiler, responsible for translating the intermediate representation into machine code. As input, the generator receives a formal specification of the target machine (architecture and instruction set).

The idea of developing code generator generators[5, 6] comes from the experience of building automatic generators for parsers and syntax directed translators. Although much more complex some important systems have been developed; for instance, BEG [8] and BURG [4, 21]. In this field, also the New Jersey machine-code toolkit[10, 11, 12] should be referred

as an important contribution. Other important work concerned with the retargeting of C compilers was discussed in [3] and [17].

3 Our Approach to Robot Programming

Today, the industrial robot programming task is done basically in two ways:

1. The programmer in charged of the task can use some modeling technique but, instead of thinking only about the problem, it is necessary to think about the robot that will run the program, and about its programming language. Both the robot and the language will limitate the specification of the problem; moreover it is not possible to reuse the same program in a different robot.

2. The programmer uses some graphical development environment, where is possible to test the program before using it in the robot. It is also possible to develop programs for different robots, but it is necessary to have a library for each robot. Even with this facilities, these tools do not solve the problem of programming the robot to interact with its environment.

However, the industrial robot programming languages did not evolved in the same manner as the computer languages. Those languages, and environments, have some drawbacks:

- The typical languages are imperative, low-level or structured. Both of them are more closed to the robot specification than to the problem, difficulting the problem modeling task and all other good practices that a correct software engineering should require.

- Each industrial robot has its own programming language, which difficulties, or even turns impossible, to reuse the source code.

The proposed approach aims at overcoming these problems; it is described diagrammatically in figure 1.

At the top, there is the problem to be solved. It will be used an adequate modeling technique, responsible for decompose the problem into simple problems, that would be easily programmed; formal models are employed to describe data structures and operations necessary to solve the elements subproblems. To describe formally the overall problem and the subproblems, a truly high level language, closed to the specification instead of the robot, should be used. Then we advocate the use of an easy-to-use compiler front-end, like Grafcet, that can interpret the specification language and generate an intermediate description for the program specified. An intermediate representation is used because the front-end must be focused on the

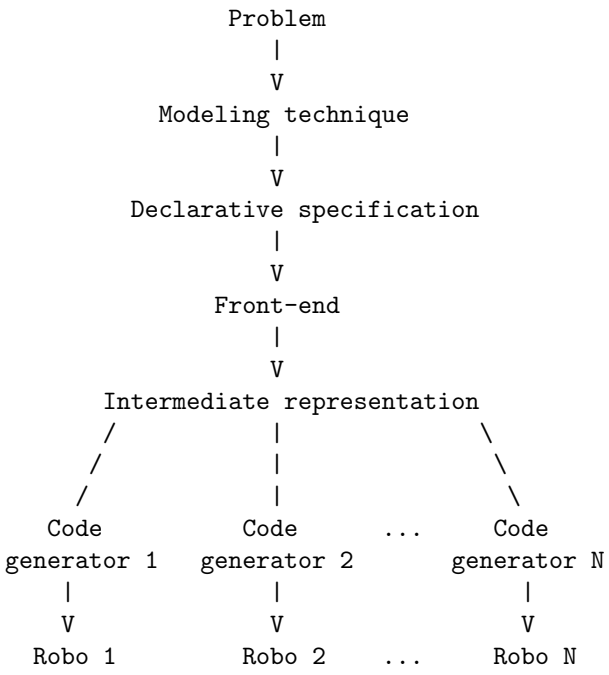

Figure 1: Proposed approach to industrial robots programming

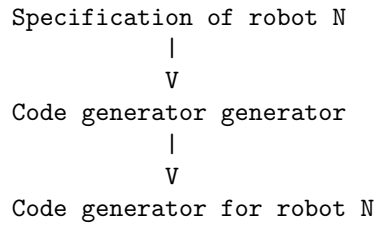

Figure 2: Automatic generator of code generators.

specification of the problem, and not on the robot. So, there will be another component responsible for translating this intermediate code into the code of a robot. Because this compiler back-end is specific for a single robot, it is necessary to have a lot of back-ends, each of them adapted to a specific target (robot's architecture and machine code). To create these code generators, it will be used an automatic generator, as can be seen in figure 2. This tool, based on the known intermediate representation, and on the robot specification, that must be included someway, will produce an optimal code generator for the specified robot.

4 Summary and Conclusions

This paper presented an interesting approach to industrial robot programming, because it covers all the stages of the programming task, from the modeling of the system until the robot code generation.

The system prototype is under construction. When this project ends, the approach will have the following features:

- It will be possible to generate code for different robots, based on the same source program;
- It will be used a high-level declarative language to specify the programs;
- It will be used a friendly front-end, to make the programming task easy;
- It will have a automatic generator for robot code generators.

The main objective of this approach is to make the programming task easy, with the possibility of reusing the source code to program different industrial robots, allowing to explore their potentialities. The programs created to control industrial robots today make them act as programmable logic controllers that can move; but there are much more things to explore. Maybe the problem arises from the low level of the programming languages available, because they do not make easy to program robots as it is to program computers, and there is no reuse of source code. Or maybe the problem resides in the simple fact that there is no actual (economic/practical) interest in such a possibility.

Acknowledgments

Gustavo Vasconcelos Arnold is professor at the Universidade Católica do Salvador, and is currently at the Universidade do Minho, aiming to get his Phd. degree, under a FCT schoolarship.

References

[1] "ABB Flexible Automation AB." *Rapid Reference Manual 3.0*

[2] G. V. Arnold. "Controle de uma Célula de Manufatura Através da Linguagem RS Estendida." *Anais do I Congresso de Lógica Aplicada à Tecnologia (LAPTEC'2000)*, pp. 145–163, 2000.

[3] C. Fraser, and D. Hanson. *"A Retargetable C Compiler: design and implementation."* Addison Wesley Publishing Company, 1995.

[4] C. Fraser, R. Henry, and T. Proebsting. "BURG - fast optimal instruction selection and tree parsing." *SIGPLAN Notices*, vol. 27, pp. 68 – 76, 1991.

[5] C. W. Fraser. *"Automatic Generation of Code Generators."* Phd Dissertation, Yale University, New Haven, 1977.

[6] C. W. Fraser, and A. L. Wendt. "Automatic Generation of Fast Optimizing Code Generators." *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 79 – 84, Atlanta, Georgia, 1988.

[7] *"O Grafcet – Diagrama Funcional para Automatismos Sequenciais."* Telemec, Portugal, 1982.

[8] H. Emmelmann, and F. W. Schroer. "BEG - a generator for efficient back ends." *Proceedings on Programming Language Design and Implementation*, vol. 24, Oregon, 1989.

[9] M. S. Mujtaba, R. Goldman, and T. Binford. "Stanford's AL Robot Programming Language." *Computers in Mechanical Engineering*, August 1982.

[10] N. Ransey, and M. Fernandez. *"New Jersey Machine-code Toolkit."* Technical Report, Departament of Computer Science, Princeton University, 1995.

[11] N. Ransey, and M. Fernandez. "The New Jersey Machine-code Toolkit." *Proceedings of The USENIX Technical Conference*, pp. 289 – 302, New Orleans, 1995.

[12] N. Ransey, and M. Fernandez. *"New Jersey Machine-code Toolkit Architecture Specifications."* Technical Report, Departament of Computer Science, Princeton University, 1996.

[13] P. J. Matos. *"Estudo e Desenvolvimento de Sistemas de Geração de Back-ends do Processo de Compilação."* Masters Dissertation, Dpto de Informática, Universidade do Minho, Braga, Portugal, 1999.

[14] R. A. Brooks. "A Robust Layered Control System for a Mobile Robot." *IEEE Journal of robotics and Automation*, Vol. RA-2, n. 1, pp. 14–23, 1986.

[15] R. E. Johnson, C. McConnell, and J. M. Lake. "The RTL System: A framework for code optimization." *Proceedings of the International Workshop on Code Generation*, pp. 255 – 274, Dagstuhl, Germany, May 1991.

[16] R. H. Taylor, P. D. Summers, and J. M. Meyer. "AML: a manufacturing language." *The International Journal of Robotics Research*, vol. 1, No 3, 1982.

[17] R. Stallman. *"Using and Porting the GNU Compiler Collection (GCC)."* iUniverse.com, Inc, 2000.

[18] S. J. Piola. *"Uso da Linguagem RS no Controle do Rob Nachi SC15F."* Trabalho de Conclusão de Curso de Graduação, Departamento de Informática, UCS, Caxias do Sul, Brasil, 1998.

[19] S. Muchnick. *"Advanced Compiler Design and Implementation."* Morgan Kaufmann Publishers, 1997.

[20] S. S. Toscani. *"RS: Uma Linguagem para Programação de Núcleos Reactivos."* Phd Dissertation, Depto de Informática, UNL, Lisboa, Portugal, 1993.

[21] T. A. Proebsting. "BURS Automata Generation." *ACM Transactions on Programming Languages and Systems*, 17(3), pp. 461 – 486, 1995.